The main function is mandatory in every program.

It returns an integer, where 0 means that the program execution finished successfully. Anything else indicates failure.

iostream is the name of a library. It defines many objects and functions, like cout and endl, which print to the standard output.

std:: denotes the namespace where cout and endl live. Different functions can share a name if they reside in different namespaces.
Control structures - if, else

```cpp
if(condition) {
    statement;
}
else if(condition) {
    statement;
}
else {
    statement;
}
```

- if evaluates the condition. If it is true, the statement is executed.
- If it is false, the statement in the optional else clause is executed.
- if and else can be nested.

```cpp
if(5 == 10) {
    std::cout << "This computer is insane" << std::endl;
}
else if(5 == 5) {
    std::cout << "Everything is fine" << std::endl;
}
else {
    std::cout << "This will never happen" << std::endl;
}
```
Control structures - for, while

The `for` and `while` loops execute statements while some condition is met. They are functionally equivalent.

- Use a `for` loop when you know how many iterations you want to do.
- Use a `while` loop when the number of iterations is unknown, for example if the stopping condition depends on user input.

```cpp
for(int i = 0; i < 10; ++i) {
    std::cout << "i equals " << i << std::endl;
}

bool keepGoing = true;
while(keepGoing) {
    std::cout << "Still going!" << std::endl;
    keepGoing = readUserInput(); //This magical function returns true or false
}
```
Control structures - continue, break

- The continue statement is used in loops to skip directly to the next iteration. It works in both for and while loops.

```cpp
for(int i = 0; i < 10; ++i) {
    if(i == 5) continue; // 5 won't be printed
    std::cout << "i equals " << i << std::endl;
}
```

- The break statement is used to exit the loop entirely. It works in for and while loops as well as switch clauses (next slide).

```cpp
while(true) {
    std::cout << "Still going!" << std::endl;
    if(readUserInput() != true) break;
}
```
Control structures - switch, do-while

- The `switch` clause can be used to replace many `if` statements.

```cpp
switch(variable) {
    case 0:
        std::cout << "variable is 0" << std::endl;
        break;

    case 1:
        std::cout << "variable is 1" << std::endl;
        break;

    default:
        std::cout << "variable is neither 0 nor 1" << std::endl;
}
```

- The `do-while` loop works like a `while` loop, except the condition is checked at the end of the loop instead of the beginning.

- This guarantees that the statement will be executed at least once.

```cpp
bool keepGoing = true;
do {
    std::cout << "Still going!" << std::endl;
    keepGoing = readUserInput(); //This magical function returns true or false
} while(keepGoing);
```
Namespaces

- A namespace is a place where variables, classes and functions live.
- They can share names as long as they live in different namespaces.
- Typing `std::` in front of all standard functions soon gets tiresome. The `using` keyword allows them to be used without a qualifier.
- If you use an entire namespace, beware of collisions (e.g. `std::count` exists).

```cpp
#include <iostream> //For cout

using std::cout; //Now we don't have to type std::cout. Just cout will do.
using namespace std; //Like the above but for everything in the std namespace

namespace first {
    int a = 10;
}

namespace second {
    int a = 20;
}

int main() {
    cout << first::a << endl; //Will print 10
    cout << second::a << endl; //Will print 20
    first::a = 30;
    std::cout << first::a << std::endl; //Will print 30. Using std:: still works.
}```
I/O - Standard input and output

- We already know how to use `std::cout` to write to standard output. To read from standard input, use `std::cin`.

```cpp
#include <iostream> //For cin and cout  

int main() {
  int userInput = 0;
  std::cin >> userInput;
  std::cout << "The user provided " << userInput << std::endl;
  return 0;
}
```

- The read operation evaluates to true if successful. A common trick is to read many times by putting it in a `while` loop.

```cpp
#include <iostream> //For cin and cout  

int main() {
  int userInput = 0;
  int sum = 0;
  while(std::cin >> userInput) {
    sum += userInput;
    std::cout << "The sum of inputs is " << sum << std::endl;
  }
  return 0;
}
```
I/O - Reading and writing files

- Reading and writing files is done using the ifstream and ofstream classes defined in the fstream library. The following program reads numbers from a file (input.txt) and prints the sum to another file (output.txt).

```cpp
#include <iostream> //For cout
#include <fstream> //For ifstream and ofstream

int main() {
    std::ifstream inFile("input.txt"); //Name of the file to read from
    if(!inFile) {
        std::cout << "Error: could not read from file input.txt" << std::endl;
        return 1; //A nonzero return value indicates failure
    }
    double variable = 0.;
    double sum = 0.;
    while(inFile >> variable) { //Read numbers until we hit the end of file
        sum += variable;
    }
    inFile.close();

    std::ofstream outFile("output.txt");
    if(!outFile) {
        std::cout << "Error: could not write to file output.txt" << std::endl;
        return 1; //A nonzero return value indicates failure
    }
    outFile << sum << std::endl;
    outFile.close();
    return 0;
}
```
Containers. Arrays

- An array is a fixed-size sequential container used extensively in C.

```cpp
#include <iostream> //For cout and cin
using namespace std;

int main() {
    const int length = 10; //The length must be known at compile time
    int arr[length]; //This array is fixed-size
    int input;
    int pos = 0; //An array doesn’t know its own size or how many elements it contains
    while(cin >> input) {
        arr[pos] = input;
        if(pos == length) break; //Remember that the array can’t grow, so this is our limit
        ++pos; //We have to keep track of the position
    }
    for(int i = 0; i < pos; ++i) cout << arr[i] << endl; //Easy to go out of range
    return 0;
}
```

- Try to avoid using arrays in C++. Use vectors instead (next slide).
  Comments to the code above contain possible pitfalls of using arrays.

- Arrays allocated on the heap are deleted with the delete[] operator.
Vectors

- A vector is a sequential container that can change size dynamically.
- It is a template class. The vector type must be defined at compile time.
- Vectors are fast at element access and insertion/removal at the end.

```cpp
#include <iostream> //For cout and cin
#include <vector>
using namespace std;

int main() {
    vector<int> vec; //Create a vector with base type int
    int input;
    while(cin >> input) vec.push_back(input); //Store each input
    for(size_t i = 0; i < vec.size(); ++i) cout << vec.at(i) << endl; //Print them back
    return 0;
}
```

- Use at to access individual elements. It’s also possible to use []. Try to avoid this! There is no bounds checking at run time. Your bugs will go unnoticed.

```cpp
vector<int> vec; //Create an empty vector
cout << vec[3] << endl; //Index is out of bounds. Your program will happily print garbage
cout << vec.at(3) << endl; //Using at produces an error at run time, exposing your bug
```
Strings

- A string is a sequence of characters, implemented by the string class.

```
#include <iostream> //For cout and cin
#include <string>
using namespace std;

int main() {
    string str("It’s dangerous to go alone, take this!");
    size_t pos = str.find("take"); //Position in string where "take" is found

    cout << str.substr(0, 18) << str.substr(pos) << endl;
    return 0; //It’s dangerous to take this!
}
```

- A C-string is a NULL terminated array of characters used extensively in C.
- Using C-strings are unwelcome for the same reasons as for arrays. Use string class instead.
- To get the C-string representation of a string, use the `c_str()` function.

```
char cString[10] = "Test"; //Contains 5 characters including the terminating \0
const char* cStringPtr = "Test"; //Pointer to string literal, can’t be modified
string cppString("Test"); //Using cppString.c_str() returns const char*
```
Command line parameters

You can pass parameters to a program via command line. They arrive as C-strings contained within an array.

The first parameter is always the name of the program. Let’s say, for the sake of example, that it’s called ‘commandLineParams’.

Here is what it would look like if built and run from a terminal.

```
$ g++ -o commandLineParams commandLineParams.cpp
$ ./commandLineParams abc 123 -bla --bla
Received 5 parameters
./commandLineParams
abc
123
-bla
--bla
```
The stack and the heap

- The memory available for a program to use (at least as far as we’re concerned) is made up of two areas - The stack and the heap.

- The stack is a small (megabytes), fixed size chunk of memory for local variables. All examples so far have used only the stack.

- When a variable on the stack falls out of scope, it is deallocated. You don’t have to worry about memory management with the stack.

- The stack is small, so it overflows if you put too many things on it. But don’t worry - This typically only happens due to bugs (e.g an infinite loop).

```c
#include "coords.h"

void makeCoordinates(int b) {
    coords co(b, b*5);
}

int main() {
    int a = 1;
    makeCoordinates(a + 2);
    //Grayed out variables have now been deallocated
    return 0;
}
```

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>co.x</td>
<td>3</td>
</tr>
<tr>
<td>co.y</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The stack and the heap

- The heap is a large pool of memory that can grow dynamically.
- To put a variable on the heap, create it with the `new` operator. This operator returns a `pointer` through which the variable is accessed.
- A pointer is really just an integer. The number corresponds to a memory address. The pointer `points` to that memory.
- Variables on the heap are never deallocated automatically. The memory must be freed manually using the `delete` operator.
- The pointer itself is on the stack and is deallocated automatically.

```cpp
#include "coords.h"

void makeCoordinates(int b) {
    coords* co = new coords(b, b*5);
}

int main() {
    int a = 1;
    makeCoordinates(a + 2);
    //Grayed out variables have now been deallocated
    return 0;
}
```
Pointers and references

- A pointer can point anywhere in memory, both the stack and the heap.
- To declare that a variable is a pointer, put an asterisk (*) after its type.
- To get the memory address of a variable, use the reference operator (&).
- If you have a pointer and you want the value that the pointer points to, use the dereference operator (*). That’s right - The asterisk has two uses!

```cpp
int foo = 10; //Two regular variables
int bar = 20;
int* p1; //Two pointers to int
int* p2;
p1 = &foo; //p1 points to foo
p2 = &bar; //p2 points to bar

*p2 = 30; //bar = 30
*p1 = *p2; //foo = bar
p1 = p2; //p1 now points to bar
*p1 = 40; //bar = 40
```

- To access members of a class via pointer, use the arrow (->) operator.

```cpp
betterCoords a(1, 1); //Regular object
a.SetCartesian(2, 2); //Access with dot
betterCoords* b = new betterCoords(1, 1); //Pointer to object
b->SetCartesian(2, 2); //Access with arrow. This is the same as (*b).SetCartesian(2, 2)
```
Pass by value, reference or pointer

- When calling a function, you are really passing \textit{copies} of all the arguments.
- If you want to change the passed values, you must use references or pointers.

```c
int x = 1;
int y = 2;

void swapByValue(x, y); //This will NOT swap the values!
void swapByReference(x, y); //This will work. Using references is recommended.
void swapByPointer(&x, &y); //This will work, but don’t use pointers unless necessary.

void swapByValue(int a, int b) { //a and b are copies of x and y
    int temp = a; //Whatever we do here has no effect on the original x and y
    a = b;
    b = temp;
}

void swapByReference(int& a, int& b) { //a and b are references to x and y
    int temp = a; //For all intents and purposes, they ARE x and y
    a = b;
    b = temp;
}

void swapByPointer(int* a, int* b) { //a and b are pointers to x and y
    int temp = *a; //Not safe - What if they are NULL pointers? Use references instead.
    *a = *b;
    *b = temp;
}
```
Writing a C++ class

- A class is a container for data and functions. This simple class stores coordinates. It has two member variables; an x and a y coordinate.
- You can create many coords, e.g (2,5) and (1,1). They have the same type but different internal states. An instance of the class is called an object.
- The constructor creates new objects. The destructor cleans up when an object is destroyed. We’ll talk more about these important functions later.

class coords { //Here I declare a class of type "coords"
    public:
    coords(int xCoord, int yCoord); //Constructor. Call to create an instance of the class.
    ~coords(); //Destructor. Gets called when an instance of the class is destroyed.

    int x; //The only two member variables are the x and y coordinates
    int y;

    private: //This class has no private members
};

coords::coords(int xCoord, int yCoord) { //Simply store the user supplied coordinates
    x = xCoord;
    y = yCoord;
}

coords::~coords() {
    //There are no special tasks to perform when destroying a set of coordinates
}
Writing a C++ class

- Let’s improve the coords class. We want the ability to change an existing coordinate. We also want the ability to work in a polar coordinate system.
- At this point we should divide the code into a header file (.h) for the class declaration and a source file (.cpp) for the implementation.
- The constructor now accepts a third argument isPolar. If it is not provided explicitly, false is used. This is called a default argument.

```cpp
#ifndef BETTERCOORDS_H
#define BETTERCOORDS_H

class betterCoords {
    public:
    betterCoords(double firstCoord, double secondCoord, bool isPolar = false);
    ~betterCoords() {};
    void setCartesian(double xCoord, double yCoord); //Set coordinates in cartesian space
    void setPolar(double rCoord, double phiCoord); //Set coordinates in polar space
    double x; //Cartesian coords
    double y;
    double r; //Polar coords
    double phi;

    private:
    void transformToCartesian(); //Helper functions to transform between coordinate systems
    void transformToPolar();
};

#endif
```

Vytautas Vislavicius
Programming for Scientists
Writing a C++ class

```cpp
#include "betterCoords.h" //Include the class declaration
#include <cmath> //For sqrt, sin, cos and atan2
using namespace std;

betterCoords::betterCoords(double firstCoord, double secondCoord, bool isPolar) {
    if(isPolar) setPolar(firstCoord, secondCoord); //The user supplied polar coordinates
    else setCartesian(firstCoord, secondCoord); //The user supplied cartesian coordinates
}

void betterCoords::setCartesian(double xCoord, double yCoord) {
    x = xCoord; //Set cartesian coordinates
    y = yCoord;
    transformToPolar(); //Then calculate the equivalent polar coordinates
}

void betterCoords::setPolar(double rCoord, double phiCoord) {
    r = rCoord; //Set polar coordinates
    phi = phiCoord;
    transformToCartesian(); //Then calculate the equivalent cartesian coordinates
}

void betterCoords::transformToCartesian() {
    x = r*cos(phi);
    y = r*sin(phi);
}

void betterCoords::transformToPolar() {
    r = sqrt(x*x + y*y);
    phi = atan2(x, y);
}
```
Constructors and Destructors

- Constructors are called to create new instances of a class. They should initialize all member variables of the class. In general they accept arguments.

```cpp
coops::coops(int xCoord, int yCoord) {
  x = xCoord; //Use the supplied values
  y = yCoord;
}
coops myCoops(2, 5); //How to invoke the constructor
```

- A constructor that takes no arguments is called a *default* constructor.
- Always write one. It is often invoked automatically, e.g. vector<coops>(5).

```cpp
coops::coops() {
  x = 0; //Choose a reasonable default value
  y = 0;
}
coops myCoops(); //Will be (0,0)
```

- To create copies of other objects, write a *copy* constructor.

```cpp
coops::coops(coops& toCopy) {
  x = toCopy.x; //Copy values from the other object
  y = toCopy.y;
}
coops myCoops(existingCoords); //Initialize as copy of existingCoords
coops myCoops = existingCoords; //These two lines are equivalent
```
Constructors and Destructors

- A class must have a non-copy constructor. If you do not write one, the compiler automatically generates a default constructor with an empty body.

- A class must have a copy constructor. If you do not write one, the compiler generates one that performs a member-wise copy (aka a shallow copy).

- Consider this line class. Note that it stores pointers to coordinates.

```cpp
class line {
    public:
    line(double x1, double y1, double x2, double y2); //Constructor
    ~line(); //Destructor
    betterCoords* start; //For some reason, we have chosen to store pointers to the coords
    betterCoords* stop;
};
```

- A shallow copy copies the pointers rather than what they point to.

```cpp
line::line(line& toCopy) { //This is a shallow copy
    start = toCopy.start;
    stop = toCopy.stop; //Changing toCopy will affect line. We don’t want this!
}
```

```cpp
line::line(line& toCopy) { //After a deep copy, line and toCopy are independent
    start = new betterCoords(toCopy.start->x, toCopy.start->y);
    stop = new betterCoords(toCopy.stop->x, toCopy.stop->y);
}
```
Constructors and Destructors

- The destructor is automatically called when an object is destroyed.
- It should delete objects on the heap and perform any other cleanup tasks.
- The compiler generates an empty destructor if you don’t write one yourself.

```cpp
line::~line() {
    if(start) { //Safeguard against deleting NULL pointers
        delete start; //This destroys the object pointed to by start
        start = 0; //Not necessary here but good practice in more complicated cases
    }
    if(stop) {
        delete stop;
        stop = 0;
    }
}
```

- Once all pointers to a heap allocated variable are lost, it can’t be deleted.
- This is called a *memory leak*. Here’s a good rule of thumb:
- Every call to `new` should be matched by exactly one call to `delete`.

```cpp
#include <stdlib.h> /* atof */
#include <unistd.h> /* usleep */
using namespace std;

int main(int argc, char* argv[]) {
  const int sizeOfAllocationInBytes = 300000000; //300 MB
  int numberOfAllocations = 10;
  double timeToSleepBetweenAllocations = 0.3 * 1000000; // microseconds
  for (int i=0; i<numberOfAllocations; i++){
      char *myChar = new char[sizeOfAllocationInBytes]; // allocate memory in heap
      for (int j=0; j<sizeOfAllocationInBytes; j++){
          myChar[j] = 'q';
      }
      usleep(timeToSleepBetweenAllocations);
      delete myChar; // <-- possible place of memory leak, if one forget to add this line
      myChar = NULL;
  }
  return 0;
}
```
Next Lecture

- Inheritance
- Polymorphism
- The const keyword
- Type Casting
- Operator overloading
- Templates
Lists, Pairs

- A list is a container with fast element insertion and removal.
- Unlike vectors, elements in a list have no absolute position. Use an iterator to loop through them. Iterators act similarly to pointers.

```cpp
#include <iostream> //For cout and cin
#include <list>
using namespace std;

int main() {
  list<int> lst; //List with base type int
  lst.push_back(10); //Insert some elements, then iterate over the list and print them
  lst.push_back(15);
  for(list<int>::iterator it = lst.begin(); it != lst.end(); ++it) cout << *it << endl;
  return 0;
}
```

- A pair is a simple container that stores two values.

```cpp
#include <iostream> //For cout and cin
#include <utility> //For pair
using namespace std;

int main() {
  pair<int, double> p(5, 3.14); //A pair of int and double
  cout << "The pair is " << p.first", " << p.second << endl;
  return 0;
}
```
A set is a container that stores unique objects. If a set already contains a certain element, adding that element again does nothing. Sets are ordered.

Adding/removing elements takes logarithmic time, which is relatively slow.

Searching also takes logarithmic time - This is as fast as a search can get!

```cpp
#include <iostream> //For cout and cin
#include <set>
using namespace std;

int main() {
    set<int> s; //Set with base type int
    s.insert(7); //Add some elements. The order in which they are added doesn’t matter.
    s.insert(1);
    s.insert(5);
    for(set<int>::iterator it = s.begin(); it != s.end(); ++it) { //Traverse with iterator
        cout << *it << endl; //Prints 1, 5, 7
    }
    if(s.count(8)) cout << "The set contains the number 8" << endl; //Search in the set
    return 0;
}
```
Maps

- A map is an associative container that stores key/value pairs. A key can not be inserted twice, but the value of an existing key can be changed.

```cpp
#include <iostream> //For cout and cin
#include <string>
#include <map>
#include <utility> //For make_pair
using namespace std;

int main() {
    map<string, int> pBook; //Map associating strings to ints. It’s a phone book!
    pBook.insert(make_pair("Reginald", 123)); //Pairs can be inserted in various ways
    pBook.insert(pair<string, int>("Marmaduke", 456));
    pBook["Bobby Floyd"] = 789;

    map<string, int>::iterator it = pBook.find("Bruce Lee"); //How to search a map
    if(it != pBook.end()) cout << it->first << " has number " << it->second << endl;
    pBook["Reginald"] = pBook["Jim Bob"]; //Beware of using [] - Jim Bob is now in the book

    for(map<string, int>::iterator it2 = pBook.begin(); it2 != pBook.end(); ++it2) {
        cout << it2->first << " - " << it2->second << endl; //Print everyone in the book
    }
    return 0;
}
```

Bobby Floyd - 789
Jim Bob - 0
Marmaduke - 456
Reginald - 0